ACORNSOFT
The choice of experience in software.

Archimedes

FORTRAN 77

GUIDE

ACORNSOFT
The choice of experience in software.

FORTRAN 77

GUIDE

First published 1988
Release 2
Published by Acorn Computers Limited
Fulbourn Road
Cambridge CB1 4JN
Part number 0481,923

CONTENTS

INTRODUCTION

FORTRAN has long been regarded as the programming language most suited to scientific and numeric applications. FORTRAN 77 is the latest standardised version of the language. This manual describes the use of Acornsoft FORTRAN 77 for the Archimedes personal workstations; note that it is not a tutorial.

The Acornsoft FORTRAN 77 compiler has been fully validated in conformance with the American National Standard Programming Language FORTRAN X3.9-1978 (ANS FORTRAN). Detailed language specifications are given in the publication *American National Standard Programming Language FORTRAN, X3.9-1978* which is available from the British Standards Institute. Less technical approaches are provided in *A Structured Approach to FORTRAN 77 Programming* by T M R Ellis, published by Addison Wesley, and *A Pocket Guide to FORTRAN 77* by Clive Page, published by Pitman.

From now on, unless otherwise stated, or made obvious from the context, FORTRAN 77 is taken to mean the Acornsoft implementation of FORTRAN 77 for the Archimedes personal workstations.

CONVENTIONS USED IN THIS MANUAL

The following conventions are used throughout this manual:

- Text entered by the user and text as it appears on the screen are shown like this:

```
This is text as it appears on the screen.
```

- Arguments to commands and options are shown as follows:

```
-debug arguments
```

The user should enter the chosen value for ***arguments***.

- Optional arguments are shown in square brackets.

SETTING UP YOUR MACHINE FOR FORTRAN

The Fortran 77 system does not run under the desktop or BASIC.

If you are using the desktop, exit by selecting the appropriate icon. You should see a prompt consisting of a single *. You can select a normal black and white colour scheme by entering BASIC and changing the screen mode:

```
*BASIC
>MODE 0
>QUIT
*
```

Alternatively, the mode may be changed at the * prompt by typing Ctrl/V followed by 0 (hold the Ctrl key down while pressing V and then press the 0 key).

If you are running BASIC, exit by using the QUIT command.

You can configure your machine so that it starts up ready for Fortran by typing:

```
*CONFIGURE LANGUAGE 0
```

When you next switch the machine on, it will start at the * prompt. The desktop may be entered by typing DESKTOP:

```
*DESKTOP
```

BASIC may be entered by typing BASIC:

```
*BASIC
```

You can configure the machine to start in the desktop again by typing:

```
*CONFIGURE LANGUAGE 3
```

or to start in BASIC by typing:

```
*CONFIGURE LANGUAGE 4
```

Fortran 77 programs require the floating point emulator to be installed. To do this on Arthur 1.2, you must load the emulator from the Fortran 77 disc. If you are not sure what version of Arthur you are using, perform the following steps to find out:

- quit the desktop
- from the * prompt, type:
  `fx0`

A message of the form

```
Arthur 1.20 (01 September 1987) (Error number &F7)
```

will be displayed. If the version number is 1.20, insert the Fortran 77 disc and type:

```
$.lib*.fpe
```

The emulator must be installed each time the machine is switched on or reset with a hard break. The installation section below shows how to set up a *!Boot* file to do this automatically.

DIRECTORY STRUCTURE

The Fortran 77 system consists of a *compiler* which converts Fortran programs to machine code, a *linker* which combines compiled programs with the Fortran *library* into executable images, and a number of *command files* which are used to run the compiler and linker. These command files assume the following directories:

f77: contains Fortran 77 source files.

aof: contains machine code files produced by the compiler.

tmp: temporary scratch files produced during the compilation process.

In addition, the directories Library and Execlib are used to hold the standard parts of the system.

INSTALLATION

Fortran 77 may be used with a variety of disc configurations. Installation instructions for the simplest cases are given below.

Before doing anything else, make a backup copy of the distribution disc and keep the original somewhere safe. To make the copy, ensure that the original is write-protected (by sliding the tab to uncover the hole) and then type:

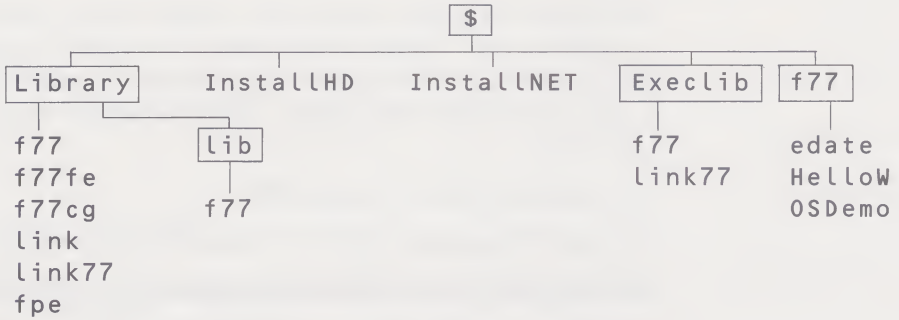
```
*backup 0 0 q
```

You will be prompted for the source and destination discs.

The Fortran 77 release disc contains the following files:

InstallHD	Utility to install system on hard disc
InstallNET	Utility to install system on network file server
Library.f77	Program for f77 command file
Library.f77fe	Fortran 77 compiler, part one
Library.f77cg	Fortran 77 compiler, part two
Library.link	Linker
Library.linkf77	Program for linkf77 command file
Library.fpe	Floating point emulator
Library.lib.f77	Fortran library
Execlib.f77	f77 command file
Execlib.linkf77	linkf77 command file
F77.edate	Example program for debugger session
F77.HelloW	Simple test program
F77.OSdemo	Demonstration program for Arthur interface routines

The following tree displays the structure of the disc more graphically:



Operating system variables

The Arthur operating system uses several *string variables* to control the running of programs. One of the most important is Run$Path, which is used as a list of directories to be searched for a program. In addition, the Fortran 77 system uses three special variables to specify the location of standard files. These are:

F77$Execlib: The directory containing the standard command files f77 and linkf77. The default if this variable is not set is $.Execlib. (the directory $.Execlib on the current disc and filing system). Note that the directory name includes a final full stop. If the command files are not on the current disc or filing system, F77$Execlib must be set so that the command processor can locate them. For example, if the files have been moved to a network file server, the variable could be set as follows:

```
*set F77$Execlib net:$.Execlib.
```

F77$Library: The full name of the directory containing the standard Fortran library. The default if the variable is not set is $.Library.lib.f77.

F77$Tmp: The directory used for temporary scratch files created during the compilation process. The default is $.Tmp. (again, note the final full stop in the value). This is suitable for most configurations. If you are using the system on a shared network file server, you must have a private tmp directory, since $.Tmp would be common to all users. In this case, you should set F77$Tmp as follows:

```
*set F77$Tmp tmp.
```

Scratch files will then be created in tmp in your current directory.

Single floppy disc

The Fortran 77 system can be used with a single 800K floppy disc. Prepare a disc by making another copy of the distribution disc. You should then delete

unwanted files (such as the installation utilities and example programs) to maximise the free space on the disc. You will also need to install a suitable editor (such as Twin) by copying it from another disc.

You will find that you have about 300K free on the disc. This is adequate for developing moderately-sized programs. You should not attempt to store too many programs on a single disc. You should also run *COMPACT occasionally, to ensure that the disc space is not badly fragmented. If you need more space, object files resulting from compilation may be copied on to another disc containing the linker.

The default settings of F77$Execlib, F77$Library and F77$Tmp are suitable for a system with a single floppy disc.

Two floppy discs

If you have a system with dual floppy drives, you can keep the Fortran 77 system on the first disc and your programs and data on the second. Make a copy of the distribution disc (using the *BACKUP command again) and use it as the system disc in drive 0. You will also need to copy a suitable editor (such as Twin) on to the disc.

Set up an empty program disc in drive 1 by creating directories f77, aof and tmp:

```
*dir :1.$
*cdir f77
*cdir aof
*cdir tmp
```

You can copy the demonstration programs from the system disc with a command like:

```
copy :0.f77.* :1.f77.*
```

The standard Fortran system variables should be set as follows:

```
*set F77$Execlib :0.$.Execlib.
*set F77$Library :0.$.Library.lib.f77
*set F77Tmp $.tmp.
```

With these settings, the command files and Fortran library will be read from the system disc in drive 0, and scratch files will be created in the subdirectory $.tmp on the current disc.

You may delete the installation utilities and example programs from the system disc if you need more space.

Hard disc

The InstallHD utility on the distribution disc may be used to set up Fortran on a hard disc system. Run it by typing:

```
:0.InstallHD
```

The system files are copied to $.Library and $.Execlib on the hard disc. The sample programs are copied to $.f77. Empty directories $.aof and $.tmp are created for use by the system.

Note that InstallHD will prompt if a file might be overwritten by the copy: if this happens with system utilities like the linker, you should check that the version on the Fortran disc is later than the one you already have. The version number of a command like the linker can be obtained by:

```
link -id
```

Ideally, you should make a backup copy of the old version before overwriting it.

The default settings of F77$Execlib, F77$Library and F77$Tmp are suitable for a hard disc system.

You must create f77 and aof directories in each subdirectory that you use for Fortran programs. For example, suppose you wish to set up a subdirectory *fourier* for the development of a fourier analysis program. This would be done as follows:

```
*cdir fourier
*dir fourier
*cdir f77
*cdir aof
```

Network

The InstallNET utility on the distribution disc may be used to install the Fortran system on a network file server. To use it, you must be logged on as a system user (usually SYST).

Note that InstallNET will prompt if a file might be overwritten by the copy – if this happens with system utilities like the linker, you should check that the version on the Fortran disc is later than the one you already have. The version number of a command like the linker can be obtained by:

```
link -id
```

Ideally, you should make a backup copy of the old version before overwriting it.

InstallNET copies the standard programs to the directory $.Arthurlib on the file server. You must include this directory in the command search path before attempting to use the system. This is done by setting the system variable Run$Path:

```
*set Run$Path ,%.,net:$.Arthurlib.
```

This would search the current directory, the current library and then $.Arthurlib on the file server. Since `net:` is included explicitly, this path will also work if your current directory is on a floppy disc. This means that you can develop your programs on a floppy and run the compiler from the network file server (as long as you are logged on).

The standard Fortran system variables should be set for network use as follows:

```
*set F77$Execlib net:$.Execlib.
*set F77$Library net:$.Arthurlib.lib.f77
*set F77$Tmp tmp.
```

With these settings, the command files and Fortran library will be read from the file server, and scratch files will be created in the subdirectory tmp.

You must create f77, aof and tmp directories in your current net (or floppy) directory before using the system.

!Boot

When installation is complete, you can prepare a !Boot file to perform standard initialisation operations whenever you turn the machine on (or whenever you log in to a file server).

The Fortran initialisation procedure should:

- Load the floating point emulator if it is not already resident.
- Set the command search path (Run$Path) if you are using a non-standard disc configuration.
- Set the Fortran operating system variables if necessary.

The required commands can be entered into a !Boot file on your boot disc. If you are using a network file server, the boot file is called !ArmBoot, rather than !Boot.

For example, if you are using a hard disc or single floppy system, then all you need do is:

```
*BUILD !Boot
 1 fpe
```

If you are setting up a network system, then the !ArmBoot file in your net directory should contain:

```
set Run$Path ,%.,net:$.Arthurlib.
set F77$Execlib net:$.Execlib.
set F77$Library net:$.Arthurlib.lib.f77
set F77$Tmp tmp.
fpe
```

On a disc system, !Boot will be run automatically when you turn your machine on if you enter the following commands:

```
*OPT 4,3
*CONFIGURE BOOT
```

On a network system, !ArmBoot will be run automatically when you log into the file server if you enter:

```
*OPT 4,3
```

If you are using a system with a single floppy drive, you will need to set up a !Boot file on each Fortran disc that you use.

If you use Fortran only rarely, you may not wish to dedicate the !Boot file on your disc for initialising the Fortran system. Instead, you can create an exec file which you obey whenever you wish to use Fortran.

CHECKING THE INSTALLATION

When you have completed the installation, you can check the system by compiling and running a simple program. Either enter a new program or copy the demonstration program f77.HelloW from the distribution disc (if you have installed on a hard disc or single floppy disc, f77.HelloW will already be there). The demonstration program F77.edate is for use in a sample ASD (Arthur Symbolic Debugger) session and has a deliberate bug – do not be surprised if the results it produces are not correct!

Compile the program by typing:

```
f77 hellow
```

You should see messages like:

```
Topexpress FORTRAN 77 front end version 1.19
Program WORLD Compiled

Total workspace used 6016

ARM FORTRAN 77 code generator version 1.62
```

```
Main program (WORLD): code 104; data 20
Total code size: 104; data size: 20
```

The version numbers may be slightly different in the programs on the distribution disc.

Link the program by typing:

```
linkf77 hellow
```

The linker should run without any messages. The linked program may be run by typing:

```
hellow
```

You should see the output:

```
Hello Fortran World
```

If there are any problems, it is possible that the installation was not completed correctly. Check the following points:

- If you get errors like 'Mistake', you are probably trying to run the Fortran system inside BASIC. Exit from BASIC by typing QUIT and try again.
- If you get 'Bad command' errors, the run path (Run$Path) may not be correct for your configuration. You can display its current value by typing:

  ```
  show Run$Path
  ```

  Note that directory names in the path must end with a full stop.
- Check that the Fortran system variables are correct. A message like

  ```
  f77: can't find command file for f77
  ```

  indicates that F77$Execlib has not been set correctly or that the files in $.Execlib have not been installed properly. A message like

  ```
  f77: can't open work file" name"
  ```

  indicates that F77$Tmp has not been set correctly or that the tmp directory (usually $.Tmp) has not been created.

  Note that the values of F77$Execlib and F77$Tmp should both end with a full stop.

MISCELLANEOUS POINTS

The two parts of the Fortran compiler can be run separately. This is explained in more detail in the next chapter, 'The Compiler'. For the HelloW program, this could be done as follows:

```
f77fe f77.hellow -to $.tmp.fcode
f77cg $.tmp.fcode -to aof.hellow
```

The first command runs part one of the compiler, which reads the source file f77.hellow and writes an intermediate form to $.tmp.fcode. The second command reads this file and writes object code to aof.hellow.

The first part of the compiler (f77fe) can be used if you just want to check for compilation errors. For example:

```
f77fe f77.prog
```

will read the program in f77.prog and display any errors it might contain. If you wish you can use the *alias* feature of the operating system to create a shorthand:

```
.set alias$fe "f77fe f77."
```

This will enable you to type:

```
fe prog
```

instead of:

```
f77fe f77.prog
```

You can add commands to set alias strings to your !Boot file.

THE COMPILER

The FORTRAN 77 compiler is made up of two parts: a front end which checks that the source code conforms to the standard, and a code generator which creates the equivalent machine code program. This is in Acorn Object Format (AOF) and is linked into an executable form using the linker program. Command files are normally used to perform a compilation and to link. There are a number of arguments which can be issued to give extra control over the compilation and allow options to be specified.

The command f77 executes a command file which runs the two parts of the compiler in sequence, and so compiles the program without the need for the user to give two separate commands.

You can write your own command files for running the Fortran compiler with a different set of arguments. If you do this, it is advisable to use a different command name and leave f77 consistent with the examples in this manual. Instructions for writing new command files are given in Appendix C.

To run the two parts of the compiler and the linker separately, the command f77fe is used for the front end, f77cg is used to generate the machine code, and link is used to link AOF files into executable programs.

COMPILATION ARGUMENTS

The f77 command compiles a single source file to object form. The format of the command line is:

```
f77 [-from] name [-object name] [-opt options]
    [-debug level] [-id]
```

where *name* is a user supplied file name, *options* is a string of one or more compiler options and *level* is a debug level name (see below). Brackets ([and]) enclose optional items. The arguments can be given in any order. Explanations of each follow:

```
-from name
```

The source file is the only argument which is not optional (although the keyword `-from` is). It specifies the name of the file which contains the code to be compiled. The file must be in directory f77.

`-opt` *`options`*

Several options are accepted by the compiler. These are given in the `-opt` argument. The options available are listed in the next section, *Compilation options*.

`-object` *`name`*

By default, the AOF output is written to the file `aof.name`, where *`name`* is the source file name. The `-object` argument can be used to direct the output to another file in the aof directory (the name given is prefixed with 'aof.' before use).

`-debug` *`level`*

The `-debug` argument controls the amount of symbolic debugging information included in the AOF file for use with ASD, the Arthur Symbolic Debugger. (Note that Appendix F contains a sample ASD Fortran session). The level should be one of the following:

`none`	No information. This is the default.
`min`	Subroutine and function names only.
`vars`	Subroutine and function names, and variable name information.
`lines`	Subroutine and function names, and line number information.
`all`	Subroutine, function, variable and line information. `max` is a synonym for `all`.

Normally, `none` is used for a working program and `all` for programs under development. The full information is quite bulky and so should be avoided when not debugging. The intermediate levels can be used to provide some debugging assistance whilst saving space.

`-id`

The `-id` (or `-identify`) argument causes the f77 command to display the version number of the FORTRAN 77 system.

The special keyword `-help` can be used to obtain a summary of the arguments expected by the `f77` command:

```
f77 -help
```

Example compiler commands

The minimal command

```
f77 Fprog
```

All source files by default reside in a subdirectory called 'f77'. This command therefore compiles the source program `f77.Fprog`. The output will be directed to `aof.Fprog`.

Error messages are sent to the VDU. Default compilation options are used.

Redirecting the object file and using compiler options

```
f77 Fprog -object Fred -opt +6
```

This compiles `f77.Fprog` producing the object file `aof.Fred` using the FORTRAN 66 option.

```
f77 prog -debug all
```

Compile `f77 prog` to AOF in `aof.prog`, with full debugging information included.

```
f77 -help
```

Display summary of arguments and options.

```
f77 prog2 -object zzz -debug min -id
```

Compile `f77.prog2` to AOF in `aof.zzz`, with minimal debugging information. The FORTRAN 77 version number is displayed.

COMPILATION OPTIONS

The `-opt` argument is followed by a list of compilation options (in upper or lower case).

The options B, H, T, 6 and 7 are enabled or disabled by preceding them with + or –. The options X, L and W must be followed by a number. The default for the full set of options is:

```
L1W2X0 -BHT6
```

This means that code generator line numbering is set to level 1; level 2 warning messages are given; there is no cross-referencing output, no bound checking, and Hollerith constants are not allowed; tracing and FORTRAN 66 are disabled.

The options have the following meanings:

B Causes the compiler to generate bounds checking code. Array or substring subscripts out of range will cause run-time errors to be reported in programs compiled with this option.

H When enabled, this option allows Hollerith constants to be used in DATA statements to initialise non-character variables (for example, INTEGER).

L *n* This option is followed by a number which indicates the level of line numbering included in the code for backtrace purposes (see the chapter entitled *Errors and debugging*). The levels available are:

- 0 no line numbering
- 1 numbers lines containing subprogram calls
- 2 statements which can cause a run-time exception
- >2 numbers every line

Higher levels cause more code to be generated. If a hardware exception occurs in a module compiled with level 1, the backtrace system will not be able to determine the exact line number; instead a range of numbers will be given (for example, 100/106). The error will lie in this range.

6 This option allows FORTRAN 66 feature to be used; if enabled, it implies the H option. When set, most constructs which have different meanings in the two versions are interpreted according to the FORTRAN 66 definition. In particular:

- DO loops will always execute at least once.
- Hollerith (nH) constants are allowed in DATA and CALL statements, and quoted constants in calls are not of CHARACTER type.
- Non-CHARACTER array names are allowed as format specifiers.

When the FORTRAN 66 switch is used, Hollerith and quoted constants are treated in the same way when used as arguments in CALLS - they are not of CHARACTER type. The option is provided for use with FORTRAN 66 programs which store character information in numeric data types.

For example, the following calls will have identical effects at run time if the FORTRAN 66 switch is used:

```
call jim ('abcd')
call jim (4habcd)
```

If the FORTRAN 66 switch is set, run-time FORMATs specifiers may also be non-CHARACTER array names.

For example:

```
double precision d(3),num
data d(1), d(3) /8h (1X,D20., 5h,I5/)/
data num /2h10/
...
d(2) = num
...
write (6, d) 2.3d0, 10
...
```

This facility was introduced to assist in the implementation of FORTRAN 66 programs; it is strongly recommended that new programs use CHARACTER formats.

T This causes the compiler to plant tracing code in the output file. The +T switch causes the front end to embed calls to special trace routines at various points in the program, such as program unit entry, DO statements, labelled executable statements and subprogram calls (see the chapter entitled *Errors and debugging*).

W*n* This sets the warning message level. A following digit of 0-4 is interpreted from the zero level as 'suppress all warnings' to 'print all warnings' (level 4). See the chapter entitled *Errors and debugging* for more details.

X*n* This is followed by a cross-reference listing width of 18 or more for maximum legibility. A value of zero suppresses cross-referencing. The upper limit depends upon the device to which the listing is being sent (for example, the printer). Cross-reference information is given immediately after the END statement of a program unit. For each name, the type is given, together with the lines on which it is referenced. For each statement label, the type (executable or non-executable) and the line number of the statement is given, as well as the lines on which the label is referenced.

7 This option is used to control warnings about the use of FORTRAN 77 language extensions. If unset, warnings are not produced; if set, messages are produced when the warning level (Wn) is 2 (the default) or greater. The option is unset by default, so that the extensions may be used without messages, whatever the warning level.

COMPILING IN SEPARATE STAGES

As an alternative to using the f77 command to execute a command file that runs both parts of the compiler, the front end and the code generator can be run separately. This section gives details of how to do this.

Front end

The front end (f77fe) reads a FORTRAN 77 source program and converts it to a special intermediate form known as FCODE.

The options handled by the front end are X, W, T, 6 and 7. The default settings are X0W2-T67.

Command format

The front end has the following command format:

```
f77fe [-from] file [-to file] [-list file]
      [-opt options] [-ver file]
```

`-from`
FORTRAN 77 source program.

`-to`
FCODE output file. If this argument is not quoted, no FCODE is produced.

`-list`
Listing file. If LIST is quoted, a listing of the source program with line numbers is sent to the file, together with any error messages. Otherwise error messages are sent to the initial output stream, and no listing is produced.

`-opt`
Front end option string. The options available were described in the section entitled *Compilation options*.

`-ver`
Output file for compiler messages and errors; if omitted, output is to the terminal.

The special keyword `-help` can be used to obtain a summary of the arguments expected by the front end:

```
f77fe -help
```

Examples

```
f77fe f77.prog -to tmp.fcode
```
Compile source program in f77.prog to FCODE in tmp.fcode.

`f77fe f77.prog -ver x`
Compile f77.prog, producing no FCODE output, with messages sent to the file x.

`f77fe f77.prog -to tmp.fcode -list list.prog`
Compile as before, but also send source listing to the file prog in directory list.

`f77fe f77.prog -to tmp.fcode -opt t`
Compile with tracing calls included.

Code generator

The code generator takes an FCODE file and produces an object file and/or assembler output.

The options handled by the code generator are L, B and H. The option 6 may be used instead of H. The default settings are L1−BH.

The front end options T, 7, W and X are ignored by the code generator, whilst the front end ignores B and L, so that the same option string may be given to both programs, if required.

Command line

The code generator has the following command format:

`f77cg [-fcode] file [-to file] [-asm file] [-ver file]`
`[-map file] [-opt options] [-debug level]`
`[-source name]`

`-fcode`*file*
FCODE input file.

`-to`*file*
Object output file. If this argument is not quoted, no object file is produced. The object file is in an AOF file, and may be linked with other AOF files using the linker Link to produce an executable file (see the chapter entitled *Using the Linker*).

`-asm`*file*
Assembler output file. If this argument is quoted, a disassembled version of the object code is sent to the file.

`-ver`*file*
Output file for code generator messages and errors; if omitted, output is to the terminal.

`-opt`*options*
Option string. The options available have already been described.

`-map`*file*
The file used for the code generator map output. The map gives the name, type and location of local and COMMON variables in each program unit. The location is relative to the start of the static area for a local variable and is the offset in the block for a COMMON variable. The offset of each statement number from the start of the code is also given.

`-debug`*level*
Level of debug information. The levels available have already been described.

`-source`*name*
Name of FORTRAN source file to be included in debugging information. Not used if the debug level is `none` (the default).

The special keyword `-help` can be used to obtain a summary of the arguments expected by the code generator:

```
f77cg -help
```

Examples

`f77cg tmp.fcode -to aof.prog`
Generate code from FCODE in tmp.fcode to an object file in aof.prog.

`f77cg tmp.prog -asm vdu:`
Code generate tmp.fcode, sending assembler output to the terminal.

`f77cg tmp.fcode -to aof.prog -map map.prog`
Code generate as before, but also send map output to map.prog.

`f77cg tmp.fcode -to aof.prog -opt +b`
Code generate with bound checking code inserted.

`f77cg tmp.fcode -to aof.prog -debug all -source f77.prog`
Code generate with full debugging information, with the FORTRAN source specified as f77.prog.

LINKING AND EXECUTION

A compiled FORTRAN program is linked using the standard Archimedes linker. The FORTRAN 77 library file should be quoted as one of the input files, using the library qualifier /l. The resulting program is run in the normal way.

Use the link command sequence as follows:

```
linkf77 source
```

The linkf77 command sequence uses value of the system variable F77$Library as the name of the FORTRAN library file. If the variable has not been set, the default name is $.library.lib.f77. The variable should be set if the library has been stored on a different disc or filing system. For example, if the library is on a network file server:

```
set F77$Library net:$.Arthurlib.lib.f77
```

The linkf77 command sequence is for simple links and may be adapted for more advanced requirements. The basic form of a FORTRAN 77 link is:

```
link aof.jim,$.library.lib.f77/l -output jim
```

The linker may be used to combine a number of FORTRAN object modules to a single executable file. The general form of the command is:

```
link aof.mod1 aof.mod2 ... $.library.lib.f77/l -output
  prog
```

Here, `mod1`, `mod2`, etc. are the object modules and `prog` is the executable image file.

EXTENSIONS TO THE STANDARD

Acorn FORTRAN 77 offers several enhancements to the standard, which are described in this chapter. To get a warning that the extensions described in this chapter have been used, use the 7 option when compiling (see the chapter entitled *The compiler*). Further extensions concerning input/output are described in the chapter entitled *Input/output*.

HEXADECIMAL CONSTANTS

Acorn FORTRAN 77 allows hexadecimal constants to be used whenever an ordinary constant is allowed. A hexadecimal constant is of the form:

```
?<type> <digits>
```

<type> is a letter, specifying the type of the constant. It must be one of I, R, D, C, L, H, or Q (for INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER and COMPLEX*16, respectively).

The <type> letter is followed by hexadecimal <digits> (0-9, A-F). There must always be an even number of digits (that is, an exact number of bytes).

The bytes in a CHARACTER hexadecimal constant are given in the order in which they are to appear in store. With other constants, the most significant byte is given first. If the type of the constant is REAL, DOUBLE PRECISION, COMPLEX or COMPLEX *16, the number of bytes must match the size of the item in store (4, 8 or 16); for INTEGER and LOGICAL constants, there may be fewer bytes. For example:

```
CHARACTER WINDOW * (*)
PARAMETER (WINDOW = ?H1C05141E0C)
J = ?I1234
```

Here, WINDOW consists of the bytes 1C 05 14 1E OC, and J is set to the decimal value 4660.

NAMING

In Acorn FORTRAN 77, all lower-case letters (except in FORMATs and character constants) are converted into upper case upon reading the source, so all statements, identifiers and so on may be in lower case. Names may be up to 255 characters long. It is worth noting, to save confusion, that there is no limit on the length of CHARACTER values.

WHILE ... ENDWHILE

This loop construct has the syntax:

```
WHILE (logical expr) DO
...
...
ENDWHILE
```

WHILE and ENDWHILE must be nested correctly, and neither statement may be used as the terminal statement of a DO–loop, or in a logical IF.

The loop is equivalent to:

```
l1 IF (.NOT. logical expr)  GOTO l2
   ...
   ...
   GOTO l1
l2
```

This form of loop is compatible with WATFIV and the Salford FTN77 compiler for PRIME computers.

DO WHILE

This loop construct has the syntax:

```
  DO n[,] WHILE (logical expr)
  ...
  ...
n ...
```

The rules regarding nesting and the terminal statement are exactly as for normal DO loops.

This form of loop is compatible with the Fujitsu and VAX/VMS FORTRAN 77 compilers.

Block DO

The syntax of DO and DO WHILE loops has been extended so that the terminal statement number may be omitted. The loop is then terminated by an END DO statement:

```
DO v = v1,v2,v3  or  DO WHILE (logical expr)
...                  ...
...                  ...
END DO               END DO
```

END DO may not be used as the terminal statement in a labelled DO loop.

This form of loop is compatible with VAX/VMS FORTRAN 77 and the new FORTRAN standard.

RANDOM NUMBER GENERATORS

Acorn FORTRAN 77 has two routines for random number generation:

```
REAL FUNCTION RND01 ()
```

returns a pseudo−random number in the range $0.0 \leqslant r < 1.0$

```
SUBROUTINE SETRND (I)
```

selects a new random sequence. If I is zero, the sequence is non−repeatable. The generator is initialised with a call to SETRND(0) so that successive runs will produce different sequences.

INCLUDE STATEMENT

An include statement allows a file containing source code to be read in by the compiler at the point where the include statement occurs. The syntax for the statement is:

```
INCLUDE 'filename'
```

Line numbers in the include file are not recorded on the object file and will therefore not appear in a backtrace; correct line numbers are shown in the program listing and error messages.

TYPE NAMES

REAL*8 may be used as an alternative to DOUBLE PRECISION. The type names LOGICAL*4, INTEGER*4, REAL*4 and COMPLEX*8 are synonyms for LOGICAL, INTEGER, REAL, and COMPLEX, respectively.

COMPLEX*16

A COMPLEX*16 value consists of pair of DOUBLE PRECISION numbers, representing the real and imaginary parts of a complex number. The rules for the use of COMPLEX*16 are the same for COMPLEX, with a few exceptions:

- Combining a COMPLEX*16 with a REAL or COMPLEX gives a COMPLEX*16 result.
- Combining a COMPLEX with a DOUBLE PRECISION gives a COMPLEX*16 result. (This combination used to be illegal.)
- A complex constant containing a double precision value is a COMPLEX*16.
- The intrinsic function DIMAG is used to extract the imaginary part of a COMPLEX*16. DCMPLX is used to convert to COMPLEX*16; it may have one or two arguments.
- The rules for storage layout and equivalencing of COMPLEX*16 are the same as for COMPLEX, except that the individual parts are DOUBLE PRECISION, rather than REAL.
- There are new specific names for intrinsic functions with COMPLEX*16 arguments; these names must be used if it is wished to use such a function as a subroutine argument. The names are:

Generic	Specific
ABS	CDABS
CONJG	DCONJG
SQRT	CDSQRT
EXP	CDEXP
LOG	CDLOG
SIN	CDSIN
COS	CDCOS

BIT MANIPULATION FUNCTIONS

There are eight intrinsic functions concerned with bit manipulation on INTEGER arguments. The functions are expanded in-line by the compiler and may also be passed as arguments in calls. The functions are:

IAND(I, J)	logical and of I and J
IOR(I, J)	logical or of I and J
IEOR(I, J)	logical exclusive or of I and J
NOT(I)	logical complement of I
ISHFT(I, J)	return I shifted left J places if J is positive or shifted right −J places if J is negative. The result is undefined if J is not in the range −32 to +32. Bits shifted out at the end are lost; zeroes are introduced at the other end.
IBSET(I, J)	return I with bit J set to one. Bit zero is the least significant bit.

	The result is undefined if J is not in the range 0–31
IBCLR(I, J)	return I with bit J set to zero.
BTEST(I, J)	test bit J of I and return a LOGICAL result – .TRUE. if the bit is set and .FALSE. if it is clear.

Note that BTEST returns a LOGICAL result; the other functions return INTEGER.

For example:

```
IF (BTEST(IX,0)) ...
```

Test to see if IX is odd.

```
I=IAND(I,?1FF)
```

Clear all but the least significant byte of I.

```
I=ISHFT(J,-24)
```

Extract the most significant byte of J.

RELAXED RULES FOR LIST-DIRECTED INPUT

When reading a complex value using list-directed (free format) input, an integer or real constant can be given – the imaginary part of the value is set to zero.

When reading a character value, if the constant:

- does not start with a quote
- is contained on a single record
- does not contain an embedded space, comma or / character
- does not start with digits followed by a *,

then the delimiting quotes may be omitted and embedded quotes are not doubled.

ARTHUR INTERFACE ROUTINES

The FORTRAN run-time library contains some simple routines to interface to the Arthur operating system. These are as follows:

OS_Byte

```
SUBROUTINE OSBYTE (IFUNC, IARG1, IARG2)
```

Perform OS_Byte function *IFUNC* with parameters *IARG1* and *IARG2*. The arguments (corresponding to the registers R0, R1 and R2) are unchanged after the call.

```
SUBROUTINE OSBYTE1 (IFUNC, IARG1, IARG2, IRES1)
```

Perform the OS_Byte function as above with the first result (from R1) passed back in the variable IRES1.

```
SUBROUTINE OSBYTE2 (IFUNC, IARG1, IARG2, IRES1, IRES2)
```

Perform the OS_Byte function as above with the results (from R1 and R2) passed back in the variables *IRES1* and *IRES2*.

Examples:

- To disable cursor editing
```
CALL OSBYTE (4, 1, 0)
```
- To return the operating system version in `INVERSION`
```
CALL OSBYTE1 (0, 1, 0, INVERSION)
```
- To read a key with a 10-second timeout, with the ASCII code going to `KEY` and the result flag to `IFLAG`.
```
CALL OSBYTE2 (?I81, MOD (1000,256), 1000/256, KEY,
+IFLAG)
```

OS_Word

```
SUBROUTINE OSWORD (ICODE, IARRAY)
```

Perform OS_Word function *ICODE* with parameter block *IARRAY* (an integer array).

Example:

- To read the current graphics cursor position:
```
INTEGER BLOCK (0:1), X, Y
...
...
CALL OSWORD (13, BLOCK)
X=IAND (BLOCK(1), ?IFFFF)
Y=ISHFT (BLOCK(1), -16)
```

OS_CLI

```
LOGICAL FUNCTION OSCLI (STRING)
CHARACTER * (*) STRING
```

OSCLI passes the string to the command intepreter for execution. The result is .TRUE. if the command succeeded and .FALSE. if it failed. In the case of an error, the code and corresponding message may be obtained by using:

```
SUBROUTINE OSGETERROR (IERRNO, ERRSTR)
CHARACTER *(*) ERRSTR
```

Return the error code and string message corresponding to the last operating system error. This should be called immediately after a failed OSCLI call (the message space is also used by the FORTRAN IO library).

For example, to display the contents of a directory:

```
      CHARACTER*80 DIR, MESS
      LOGICAL OSCLI
      ...
      ...
      IF (.NOT. OSCLI ('CAT ' // DIR)) THEN
        CALL OSGETERROR (IERR, MESS)
        PRINT 100, IERR, MESS
100     FORMAT ('Error ', I10, ': ', A)
      ENDIF
```

INPUT/OUTPUT

This chapter describes how FORTRAN 77 input and output functions are implemented and how this affects programs.

UNIT NUMBERS AND FILES

A FORTRAN unit number is a means of referring to a file. Unit numbers in the range 1 to 60 may be used, as well as the two * units for the keyboard and screen. Unit number zero is equivalent to the asterisk units and may be used in sequential READs and WRITEs only. Note that your filing system limits the number of files you can open simultaneously – consult your filing system manual.

A unit number may be connected to an external file either by means of an OPEN statement or by assignments on the command line when the program is run. If an OPEN statement with the FILE= specifier is used, then the unit is connected to the given filename; otherwise, the command line parameters are scanned.

The format of the command line is:

command {file*} {unit=file*}

that is, an optional list of filenames followed by an optional list of assignments of a particular unit to a named file. The initial series of unkeyed filenames are connected to units 1, 2, 3 and so on. Each keyed file is connected to the given unit number. All unkeyed definitions must precede any keyed definitions.

Examples are:

```
PROG ABC DEF
```

This associates the file ABC with unit 1 and DEF with unit 2.

```
PROG 10=FILE
```

This associates the file FILE with unit 10.

```
PROG DATA 32=DATA 3=X
```

This associates DATA with unit 1, data with unit 32, and x with unit 3.

The two * units always refer to the screen and the keyboard. Any units which are not connected to a file in an OPEN statement or command line assignment also refer to these streams.

The screen and keyboard streams can be redirected to (from) a file using the standard Arthur syntax ({ > *filename* } and { < *filename* }).

A file accessed with STATUS=SCRATCH (OPEN) or STATUS=DELETE (CLOSE) is deleted when the unit is closed.

All files are closed automatically when a program terminates.

When writing to a sequential formatted file, a distinction is made between files which are to be printed and those which are not. In the former case, the first character of each record is taken as a carriage control and does not form part of the data in the record. Since any file may eventually be printed, some means is required in FORTRAN for specifying whether a given unit is to be treated as a printer. This may be done in one of two ways:

- All units in the range 50–60 assume printer output by default. On other units, the FORM='PRINTER' option can be used to select printer operation.

- Quoting FORM='PRINTER' in the first OPEN statement for the unit causes printer output to be assumed for that unit. (Note: this is an extension to the standard.)

Note that printer output does not imply output to any physical printer which may be connected to the machine.

The carriage control characters which are recognised and their representation in files are described in the section entitled *Formatted IO*.

SEQUENTIAL FILES

Opens and closes

An OPEN statement for a sequential file does not specify the direction of transfer that is required, so the actual system open operation cannot be done until the first READ or WRITE statement following the OPEN. For this reason, an OPEN statement which refers to a nonexistent file will not fail – the error will occur when a READ or WRITE is attempted, but may then be trapped by use of an ERR= specifier.

A sequential unit may be used without an explicit OPEN operation, in which case the file is actually opened on the first READ or WRITE which refers to the unit. The following subroutine is an example of the use of OPEN and ERR=. The routine copies a named file to the terminal, using unit 10.

```
      SUBROUTINE COPY (FILE)
      CHARACTER FILE* (*), LINE*72
      OPEN (10, FILE=FILE, ERR=100)
1     READ (10, '(A)', END=100, ERR=100) LINE
      PRINT '(A)', LINE
      GOTO 1
```

```
100  CLOSE (10)
     END
```

Formatted IO

Formatted (and list-directed) reads and writes are permitted on all files.

A formatted READ statement causes one or more records to be read from the file or terminal. All input records are assumed to be extended indefinitely with spaces, so that an input format may refer to more characters than are actually present in the record. Input from the terminal uses the normal line editing conventions (including cursor copying). CTRL/D is treated as end of file, which may be trapped by an END= specifier in a READ statement.

For file input, the characters carriage return (OD) and line feed (OA) are each recognised as record terminators. Form feed (OC) characters are ignored. If the record contains more than 512 data characters, the rest are ignored. The combination carriage return-line feed (or line feed-carriage return) is treated as a single record terminator.

When writing a record to a file or terminal, the carriage control characters are output first, followed by the data in the record. Trailing spaces are removed from all output records.

The following carriage control characters are recognised:

space	performs a line feed (LF)
O	performs LF/LF (extra blank line)
1	performs CR/FF (newpage)
+	performs CR (overprint)
*	no action taken

The initial LF (space/0) or CR (1/+) is not output before the first record in the file. When a file is closed, a line feed character is output if the final record contained any data characters. This is done automatically for all open files when a program terminates normally.

When writing to a non-printer unit, each record is terminated with a newline. If a prompt line is required on output, a $ (or \) character may be included in the format. This suppresses the final newline. In addition, trailing spaces are not removed from the final line output by the format. This facility may be used to generate interactive prompts, as in:

```
WRITE (6, '(A$)') 'Type an integer: '
```

The $ (or \) acts as a normal item (like /) and can occur anywhere in the format (except after any unused editing codes, since these will be skipped by the format processor).

The following example program illustrates interaction with a terminal file:

```
1 PRINT '($a)', '?'
  READ (*,*, END=3) I
  WRITE (*, 2) I, I*I
2 FORMAT (2I10)
  GOTO 1
3 END
```

The CHAR function may be used to construct bytes for output as VDU control codes. For example, the following statements will switch the screen to MODE 3 on your machine:

```
  WRITE (*, 3) CHAR(22), CHAR(3)
3 FORMAT ($,2A)
```

Note the use of the $ format descriptor to suppress the final newline.

During formatted input of numeric values, blanks are either ignored or treated as zeros, depending on the use of the BZ and BN format specifiers, and the BLANK status of the unit. All preconnected units (that is, those opened without explicit use of OPEN) have BLANK=ZERO as the default status; any unit connected by an OPEN statement has BLANK=NULL as the default. The difference in the defaults was introduced for compatibility with FORTRAN 66 and the FORTRAN 77 subset language (in FORTRAN 66, blanks are always treated as zeros).

Unformatted IO

Unformatted reads and writes are permitted on disc files only. Unformatted and formatted operations may not be mixed on any unit, unless the unit is closed and reopened.

Each unformatted WRITE statement writes a single record to the file. The record may be read back later by any READ which quotes the same number of, or fewer, variables. For example, in:

```
WRITE (1) 1, 2, 3, 4, 5
WRITE (1) 6, 7, 8
REWIND 1
READ (1) I
READ (1) J
```

i is to 1 and j to 6. The first record contains 20 bytes of data, and the second 12 bytes.

The desired effect could be achieved by padding all unformatted records to the same length, but this would lead to wasted file space in many cases. The system includes a record length before every unformatted record when it is output, and always reads the right amount when the record is read again.

The actual format of the length is the characters UF, followed by a four-byte byte count giving the number of data characters following. The UF bytes are used as a check that the file contains valid unformatted records. For example, the two records written in the example above would contain the following bytes:

```
55 46 14 00 00 00
01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00
55 46 0C 00 00 00
06 00 00 00 07 00 00 00 08 00 00 00
```

DIRECT ACCESS FILES

A direct access file consists of a number of records, all of the same length, which may be read and written in any order. The records are either all formatted or all unformatted.

An OPEN statement, quoting the record length, is always required when using a direct access file. The record length is measured in bytes, and formatted records are padded to this length with spaces.

A direct access file starts with six special bytes which identify it and give the record length. These bytes are the characters DA followed by the record length as a four-byte value (LS byte first).

It is permissible to OPEN a direct access file quoting a smaller record length than was given when the file was created.

The maximum permitted record length in a formatted direct access OPEN is 512 bytes; there is no limit for unformatted files.

If the file has been opened for updating or input, the first six bytes of the file are read and checked. The OPEN will fail if these bytes are invalid, or the specified record length is greater than the value used when the file was created.

Since it is possible both to read and write to a direct access file, the system open operation may be performed as part of the OPEN statement, rather than

being delayed to the next READ or WRITE, as is the case with sequential OPENs. Therefore, any errors which occur in the open may be trapped by an ERR= specifier in the OPEN statement.

Note that a direct access OPEN may refer to an existing file only if it is of the correct format; however, it would be simple to provide a utility program to create a new direct access file of a given size and record length. The following is an example program which uses direct access to write and read a file on unit 42:

```
      OPEN (42, ACCESS='DIRECT', FILE='DAFILE', RECL=16,
     +ERR=100, IOSTAT=IERR)
      DO 1 J = 20,1,-1
1     WRITE (42, REC=J) J, J+1, J*J, J-1
      DO 2 J=1,10
      READ (42, REC=J), K, L, M
2     WRITE (*, 3) K, L, M
3     FORMAT (1X, 3I5)
      STOP
100   PRINT *, 'OPEN FAIL:', IERR
      END
```

Note that unformatted records are the default for direct access files. The file 'dafile' used in the above example need not exist already, but if it does, it must be a valid direct access file with a record length not less than 16.

OPEN and CLOSE

The OPEN and CLOSE statements have been discussed above. The NEW and OLD values for the STATUS specifier in the OPEN statement are ignored.

INQUIRE

INQUIRE by unit

An INQUIRE by unit operation gives information on a particular unit. The EXIST specifier variable is set to .TRUE. if the unit is in the valid range. It is impossible to give accurate responses to the SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED specifiers, so 'YES' is returned if the unit is actually being used for the relevant access type, and 'UNKNOWN' is returned otherwise. Note that a unit is NAMED only if a FILE specifier was

quoted in the OPEN statement for the unit. Command line file assignments are not available to INQUIRE.

INQUIRE by file

An INQUIRE by file operation gives information on a particular filename. If the file has been quoted in an OPEN statement for a unit (and not CLOSEd), information deduced from that connection is returned (for example, DIRECT is set to 'YES' if the file is open for direct access), and the file is assumed to exist. Otherwise, if the file exists, the EXIST reply is .TRUE. and the responses to the SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED specifiers are 'UNKNOWN'.

BACKSPACE

BACKSPACE is not implemented.

ENDFILE

The operation of ENDFILE is entirely internal to the run-time system; the only effect is to set end of file status and forbid further access to the file.

REWIND

REWIND is implemented as a CLOSE followed by an OPEN. After executing a REWIND, the file is in a similar state to that arising after an OPEN statement - the system open operation is awaiting the next READ or WRITE statement.

FORMAT DECODING

Format specifications are decoded in a rather more liberal manner than implied by the FORTRAN standard.

Lower case letters

Lower case can be used instead of upper case everywhere; cases are distinguished only in quoted strings and nH descriptors, and in the `D`, `E` and `G` edit descriptors (see below).

Extraneous repeat counts

Unexpected repeat counts are ignored – that is, before `'`, `T`, `/`, `:`, `S` and `B` edit descriptors, before the sign of a `P` edit descriptor, or before a comma or closing parenthesis.

Edit descriptor separators

A comma may be omitted except where the omission would cause ambiguity or a change in meaning – thus, it cannot be omitted between a repeatable edit descriptor (such as `I5`) and an nH edit descriptor (such as `11Habcdefghijk`).

Numeric edit descriptors

As well as the standard forms `Iw`, `Iw.m`, `Fw.d`, `Ew.d`, `Ew.dEe`, `Dw.d`, `Gw.d` and `Gw.dEe`, additional forms are: `Fw`, `Dw.dDe`, `Gw.dDe`, `Dw.dEe`, `Ew.dDe` `Zw`, and `Z`.

When the exponent field width is specified, the letter used to introduce it is used in the output form (in the same case). If no exponent field width is specified then, except for G edit descriptors, the initial character of the descriptor is used in the output form (again, in the same case).

If an exponent field width is given as zero, 2 is assumed; if on output the given exponent field width is just too small for the exponent, the character introducing the exponent field is suppressed.

The Z edit descriptor provides input and output of numeric data in hexadecimal form. A field width of zero implies the correct width for the data type being transferred; Z by itself is a shorthand for Z0.

A editing

The `A` edit descriptor can also handle numeric list items; the effects are as recommended in Appendix C (Hollerith) of the FORTRAN 77 standard. If the field width is zero, the system will automatically use the right value for the data type being transferred (4 or 8).

It must be emphasised that this use of `A` editing was introduced solely to aid in the transfer of FORTRAN 66 programs: it should not be used otherwise.

Abbreviations and synonyms

symbol	*abbreviation*
OP	P
1X	X
T1	T
TL1	TL
TR1	TR
A0	A

Transfer of numeric items

The I edit descriptor can be used to transfer real and double precision values; F, E, D and G can be used to output an integer value. Note that the external form of a value that is to be transferred to an INTEGER list item must not have a fractional part or a negative exponent.

$ and \ descriptors

A $ or \ descriptor in a format suppresses the final newline when writing to a non-printer file – see the example earlier.

GRAPHICS

FORTRAN programs can use the full range of Arthur graphics facilities by writing control codes to the VDU drivers. The CHAR function is used to convert an integer code to a character for output.

The basic form of a WRITE statement to generate graphics is:

```
WRITE(*,'($,10A)') CHAR(code1),CHAR(code 2),...
```

or

```
PRINT '($,10A)',CHAR(code1),CHAR(code2),...
```

This example uses the standard asterisk output unit; any *non-printer* unit (1-49) could be used instead. The repeat count in the format (10 in this example) must not be less than the number of VDU codes in the list. The $ format descriptor is used to suppress the final newline.

The format can be given as a character constant, as above, or in a separate statement:

```
      PRINT 100,CHAR(code1),CHAR(code2),...
  100 FORMAT($,10A)
```

For example, to change to mode 12:

```
PRINT '($,2A)', CHAR(22), CHAR(12)
```

Or to change the palette for colour 1 to refer to colour 6:

```
PRINT '($,6A)', CHAR(19), CHAR(1), CHAR(6),
+               CHAR(0), CHAR(0). CHAR(0)
```

Most move and draw operations take a pair of 16-bit coordinates. These should be output as a pair of bytes. For example, the following subroutine provides an interface to the general PLOT command (VDU code 25):

```
SUBROUTINE PLOT(TYPE, X, Y)
INTEGER TYPE, X, Y
PRINT ' ($,6A)', CHAR(25), CHAR(TYPE),
+       CHAR(IAND(X,255)), CHAR(ISHFT(X,-8)),
+       CHAR(IAND(Y,255)), CHAR(ISHFT(Y,-8))
END
```

MOVE and DRAW operations are then type 4 and 5 PLOT calls.

As a simpler alternative to writing the coordinates as four separate bytes, they can be output directly as characters, using the extension which allows non-character data to be output with the A format descriptor:

```
PRINT '($,2A,2A2)', CHAR(25), CHAR(TYPE), X, Y
```

The A2 format descriptors cause the least significant two bytes of X and Y to be output as characters. The complete PLOT routine is then:

```
SUBROUTINE PLOT(TYPE, X, Y)
INTEGER TYPE, X, Y
PRINT '($,2A,2A2)', CHAR(25), CHAR(TYPE), X, Y
END
```

ERRORS AND DEBUGGING

In most cases, mistakes in a program are trapped, and indication is given as to the likely cause of the problem via error messages. Errors can be detected both by the compiler and by the run-time library. (An example of a fault which is not caught by the compiler, but by the FORTRAN run-time library, is attempting to divide by zero.) More usually, error messages are sent from the compiler. This may also generate warning messages which indicate to the programmer that the program may not behave as anticipated – for example using, but not declaring, a variable.

FRONT END ERROR MESSAGES

As mentioned in the chapter entitled *The Compiler*, the compiler is in two parts. Errors trapped by the front end are of a different type from those reported by the code generator. Front end error messages are short, obvious statements indicating that the compiler has spotted an unacceptable syntactic mistake. Since these messages are self-explanatory, they are not enunciated in great detail here. They are divided into two classes:

Class 1 errors cause the front end to abandon compilation of the current statement. The statement is printed as part of the error message, together with the number of the line on which the fault appeared, an error number, and a description of the error itself. Thus, if line 211 contained the faulty FORTRAN statement:

```
100  ERRONEOUS
```

then the message produced might be:

```
  211  100  ERRONEOUS
L  211-------?
Error (code 2311): Statement not recognised
```

Class 2 errors may be less obvious in their report of a fault and do not always refer to the line which contains the code which instigated the error. For instance, information about missing labels is given at the end of the program unit, rather than where the non-existent label was called.

The distinction between these two types of error message has been made in order to reinforce the notion that errors do not necessarily occur at the line where the message is given; careful thought and a little imagination are often needed to pinpoint the cause of some persistent error messages.

WARNING MESSAGES

The `W` compilation option enables the compiler to give advice in the form of warnings: see the chapter entitled *The compiler* for more details on its use. These warning messages are graded in severity from 1 (the most serious) to 4, and are useful in detecting areas which may cause the program to behave in unexpected ways.

Level 1 is the most serious, indicating faults such as having a statement that cannot be reached because it is unlabelled and follows a jump. Level 2 flags the use of extensions to standard FORTRAN 77 that are a potential source of trouble (for example, when moving software to another machine). Levels 3 and 4 are used to indicate items that are legal but in poor style, and thus possibly mistakes. The strict FORTRAN 77 option 7 is used to control warnings about language extensions. If unset, warnings are not produced; otherwise, messages are produced if the warning level (Wn) is 2 (the default) or greater. The 7 option is unset by default so that the extensions may be used without messages, whatever the warning level.

CODE GENERATOR ERROR MESSAGES

Certain compile-time errors cannot be detected by the front end, but are reported by the code generator. As these are not always as explicit as front end error messages, they are listed in Appendix A with a brief explanation of their most likely meaning. The same caveat applies to the interpretation of code generator error messages as applies to that of some front end error messages: the error which is reported and its line number may not directly correspond to an error in the program. For example, an array might be declared which is too big for the memory. Quite often, one error may spark off the detection of many others later on in the program. See *Appendix A* for a list of code-generator error messages.

CODE GENERATOR LIMITS

The code generator has certain internal limits on the complexity of each program unit. These are:

code size	2 Mbytes
number of labels	4096
number of local variables	8192
number of constants	8192
number of COMMON blocks	2048
number of external symbols	2048

These limits should never be exceeded in practice; it is likely that the code generator will run out of store before this happens.

RUN-TIME ERRORS

Sometimes, a program compiles correctly and links without a problem — yet when an attempt is made to run the program, an error message is produced. These error messages come from the FORTRAN run-time library and take the following form:

```
++++ ERROR N: text
```

followed by a backtrace.

`N` is an error number and `text` is a sentence describing the error. A backtrace is, as the name implies, a re-tracing of the steps which the FORTRAN run-time library has taken in attempting to run the program; each line of the backtrace output gives the name of a program unit, the addresses of the corresponding static data area and the line number. The data area address may be used in conjunction with the storage map produced by the code generator to examine the values of local variables. The address of the data area is given in hexadecimal. Note that a name in a backtrace refers to the main entry point of the program unit, and so may not be the actual name used in a call.

Example run-time error message and backtrace

```
++++ ERROR 1025: LD input data not INTEGER
Routine          data area          line
F77_INIT         &000100D8
F77_IO67         &00010000
ERR2             &0000FF04            16
ERR1             &0000F9B4            10
F77_MAIN         &0000F9B0             6
```

In this example, the main program (with default name) has called ERR1, which has called ERR2, which has attempted to read an integer using list-directed input (the top two names are internal routines in the run-time library).

The call to ERR1 in the main program was on line 6; the call to ERR2 in ERR1 was on line 10, and so on. The appearance of line numbers in the

backtrace is controlled by the compiler L option; level 1 is the default. See the chapter entitled *The compiler* for details about compilation options.

If a hardware trap occurs in a program compiled with line number level 1, it may not be possible to determine the exact line number. This is illustrated by the following trace:

```
++++ ERROR 3000: hardware trap

Routine          data area          line

ABC              &00005514          5/16
F77_MAIN         &000054EC             3
```

Here, the main program called ABC failed with a hardware trap between the lines 5 and 16 inclusive. If the program is recompiled with line level 2, the exact number will be displayed.

Code 1000 errors

There are a number of simple run-time errors producing error messages which have an error number of 1000. (An example of a code 1000 message was given in the previous section.) See *Appendix B* for a comprehensive list.

ARRAY AND SUBSTRING ERRORS

There are two errors which may be produced from a program unit which has been compiled with the bound checking option (see the chapter entitled *The compiler*):

```
++++ ERROR 1050: array bound error
```
An illegal array subscript has been used.

```
++++ ERROR 1051: substring bound error
```
An illegal substring has been used.

INPUT/OUTPUT ERRORS

Input/output errors are those which may be trapped by use of the END= and ERR= specifiers in FORTRAN 77 statements. If these are not used, an error message and code are produced as described below; otherwise, execution continues, with the error code available by use of the IOSTAT specifier.

All the messages have the general form:

```
++++ ERROR N: PREFIX UNIT - reason
```

`N` is the error code; `PREFIX` describes the IO operation being attempted (which may be OPEN, CLOSE, BACKSPACE, ENDFILE, REWIND, or READ/WRITE) and `UNIT` is the unit number, with `*` given for one of the asterisk units and 'internal' for an internal file. The rest of the message gives more information about the error.

End of file on input may be trapped with the END= specifier. The IOSTAT value in this case is −1. If END= is not used, then the message `end of file` is produced, with code 1000. Other errors may be trapped with the ERR=specifier. The IOSTAT value is the corresponding error code, as listed in *Appendix B*.

TRACING

Tracing a program's execution is a very useful debugging technique, applicable when a program compiles and runs successfully but produces unexpected output. The user selects the T option when compiling (see the chapter entitled *The compiler*) to specify that calls to special trace routines are to be included in the code. These routines will cause trace information to be output when:

- entering the program unit
- leaving the program unit
- a labelled statement is about to be executed
- the THEN clause of an IF...THEN or ELSEIF...THEN construct is about to be executed
- the ELSE clause of an IF...THEN or ELSEIF...THEN construct is about to be executed
- a DO statement is about to be executed
- another subprogram unit is about to be invoked.

The trace routines will output a message which starts with ***T and indicates the type of trace point encountered; for some of these it will also indicate a count (modulo 32768) of the number of times this trace point has been met. A special routine called TRACE can be called with a single LOGICAL argument to turn this tracing information on and off. Note that even if the trace output is off, the counting will still be done so the values produced will be correct if tracing is turned on again.

If the main program is compiled with tracing on, the user will be asked if trace output is to be produced or suppressed. If the main program is compiled without tracing, then trace output is initially enabled.

In addition to the TRACE routine, two further subroutines are supplied as part of the tracing package. The first of these, HISTOR (short for HISTORY), causes information to be output about the last few traced subprogram calls.

Each line of history information consists of a name, which may be preceded by >or by<. A right arrow indicates a traced call of a subprogram, a left arrow indicates a traced exit from a program unit, and a line with neither type of arrow indicates a traced entry to a program unit. Note that the name given when tracing entry and exit from a program unit is the name of the program unit itself rather than the name of the entry called by the user.

The final routine provided is BACKTR (short for BACKTRACE) which outputs information on the current nesting of program unit calls. The routine should be given a single logical argument; if this is TRUE then the HISTOR subroutine is involved after the backtrace information has been produced. In the Archimedes system, all tracing output is sent to the terminal or may be sent to a file using the SPOOL command.

USING THE LINKER

The Linker is an essential program for anyone developing programs in a high-level compiled language on Archimedes personal workstations. Its purpose is to combine the contents of one or more object files (the output of a compiler or Assembler) with one or more library files, producing a final executable program.

Syntax

The format of the Link command is:

`Link -output` *`file`* `[`*`options`*`]` *`files`*

The *`files`* argument is a list of input files; this is described below. `-output` is the only compulsory keyword.

Below is a list of the command line options that the Linker can take. Most of these will only be used occasionally. In the descriptions below, the important, frequently-used options are given first, followed by the less common ones. As usual, capitals are used to denote the alternative shortened form of the keyword.

`-Output`	Name of the linked output file
`-VIA`	Use a file to obtain (further) input file names
`-Case`	Make matching of symbols case insensitive
`-Base`	Set base address for output file
`-Verbose`	Print messages indicating progress of the link operation
`-Relocatable`	Generate relocatable output file
`-Dbug`	Generate an AOF image for use with the Dbug program

Notes

- the keyword `-base` is followed by a numeric argument. You can use the prefix `&` to specify hexadecimal, and the suffixes `k` for `2^10` and `m` for `2^20`.
- the default base address for the output file is `&8000` (32K). If `-dbug` is specified, the default base address is `&50000` (ie 320K).
- The item *`files`* above is a list of one or more filenames, separated by spaces. This part of the command must be given. Each of the files in the list must be in Acorn Object Format (compiled files) or Acorn Library Format (libraries). They may contain references to external objects (procedures and variables) which the Linker will attempt to resolve by matching them against definitions found in other files.

- You can use wildcards in the filename list. Names using wildcards will be expanded into the list of files matching the specification. For example, the name `aof.bas*` might yield `o.basmain, aof.basexpr, aof.bascmd`.
- Usually, at least one library file will be specified in the list. A library is just a collection of AOF files stored in a single Acorn Library Format file. You can call the procedures in the library for one language from programs written in another, as long as both languages conform to the ARM Procedure Calling Standard and both run-time libraries use the common run-time kernel. For example, an assembler program could use the C `printf` function, as long as the C run-time system had been initialised, through the common run-time kernel.
- Libraries differ from object files in the way the Linker uses them. Object files' symbols are scanned only once when the Linker attempts to resolve external references. Libraries are scanned as many times as necessary. If a required symbol is found in one of the library's component files, the whole component is incorporated into the output file.
- Two common errors given during a link are caused by unresolved and multiple references. In the first case, a symbol has been referenced from a file (whose name is given in the error), but there is no corresponding definition for the symbol. This is usually caused by the omission of a required object or library file from the list, or the mis-spelling of a symbol in the original source program.
- The second error is caused by a clash of names. For example, a procedure might have been defined with the same name as a library procedure, or as a procedure in another object file. The version of the procedure used in any situation is the one local to the reference to it.
- The `-output` keyword is obligatory. It is followed by the name of the file to which the final linked program should be written. If you just want to use the Linker to check object files for unresolved references, you can specify the device `null:` as the output file; the final object code will be discarded. The output is usually in Arthur Image Format, which can be executed directly. Alternative formats allow low-level debugging with Dbug, which forms part of the Software Developer's Toolbox.

Simple examples

Before we move on to describe the rest of the `Link` command's options, we give some examples using the syntax described so far:

```
Link -OUTPUT p.sieve aof.sieve, ansilib
Link -o %.mybasic aof.bas* lib.f77
Link -o null: aof.comp*
```

VIA KEYWORD

Sometimes you may want to link a large number of input files which would be tedious to type on a command line, and whose names can't conveniently be matched by a wildcard specification. Using the `-via` keyword, you can store a list of input filenames in another file and use this to access them. For example, suppose you created the file `basfiles` with the contents:

```
aof.main
aof.expr
aof.cmd
aof.stmnt
aof.lex
aof.filing
aof.tokens
```

If you then used the command:

```
*link -o basic -via basfiles lib
```

then the files listed in `basfiles` would be linked, together with the AOF file `lib`.

CASE KEYWORD

If you specify `-case` in the command line, then the Linker will not treat the case of letters as significant in identifiers. By default, the identifiers `main` and `Main` refer to different objects, as they are spelt differently. However, with `-case` set, they are the same identifier.

BASE KEYWORD

By default, the base address of the output file of the Linker is `&8000`. This corresponds to the start of application workspace on the Archimedes computer. Alternatively, if the `-dbug` option is given, the base address is set to `&50000`. This is so that the debugger program Dbug can load at `&8000` as a normal application, and load the file to be debugged above itself. (There are other changes when `-dbug` is given, as described below.)

Using the `-base` keyword, you can set the base address of the output file to any desired value. For example, you may want a program to have a high load address (as with the `-dbug` option set), but still be directly executable (which a dbug file in AOF format isn't).

The keyword is followed by a number giving the base address desired for the output file, eg `-base &80000`, `-base 256k` etc. When this is done, all relocatable objects in the input files are relocated using that base instead of the default.

VERBOSE KEYWORD

If you specify `-verbose` on the command line, the Linker gives a report of its progress. A message is printed as each file is opened and as each module is being relocated. For example:

```
link: opening p.basic
link: opening aof.bas1
link: opening aof.bas2
link: relocating module aof.bas1
link: relocating module aof.bas2
link: relocating module ansilib(fpprintf)
...
```

RELOCATABLE KEYWORD

Usually, when an image file is produced, it will execute correctly only at the base address given (or the default). This is because the object program will contain references to absolute addresses within the data area. However, if you specify the `-relocatable` option, the final program will be relocatable. That is, it can be loaded and executed at any address.

This feat is achieved by adding a relocation table and a small program to perform the relocation to the final object code. The relocation table is a list of offsets from the start of the program to words which need relocating. These words are adjusted by the difference between the base address of the program and the address where it was loaded. Once the relocation has been performed, the program proper starts executing.

The relocation process is very fast, and once it has been performed, the space occupied by the table is available as part of the program's heap space when it starts executing.

Note that although this ability can be used to make a program statically relocatable, it does not confer true position-independence on the program. That is, the program could not be moved in memory once it has started and still be expected to work.

DEBUG KEYWORD

If a program is linked using the `-dbug` keyword, an executable image is not formed. Instead, an AOF file is created which contains all of the symbols found in the original source files. The code segment of the file can be executed under the control of a Dbug program, and the contents of the code and data segments may be examined (and altered in the case of the data segment).

PREDEFINED LINKER SYMBOLS

There are several symbols which the Linker knows about independently of any of its input files. These start with the string `Image$$` and, along with all other external names containing `$$`, are reserved by Acorn.

The symbols are:

`Image$$RO$$Base`	Address of the start of the read-only (program) area
`Image$$RO$$Limit`	Address of the byte beyond the end of program area
`Image$$ZI$$Base`	Address of the start of run-time zero-initialised area
`Image$$ZI$$Limit`	Address of the byte beyond the zero-initialised area
`Image$$RW$$Base`	Address of the start of the read/write (data) area
`Image$$RW$$Limit`	Address of the byte beyond the end of the data area

Although it will often be the case, Acorn does not guarantee that the end of the read-only area corresponds to the start of the read/write area. You should therefore not rely on this being true.

Note also that programs can reside in read/write areas, as they sometimes contain local writeable data (eg modifiable code), and it is possible to have read-only data (eg floating-point constants and string literals in C).

These symbols can be imported as relocatable addresses by assembly language routines that might need them.

The Linker joins all areas (from all input files) with the same name and attributes together to form a single area. It then creates the two symbols *name*`$$Base` and *name*`$$Limit` to mark the start and end of the area. It is an error for two areas to have the same name but different attributes.

APPENDIX A

CODE GENERATOR ERROR MESSAGES

`argument out of range for CHAR`
The intrinsic function CHAR has been used with a constant argument outside the range 0-255.

`local data area too large`
The size of the local storage area for the program unit exceeds memory size.

`array <name> has invalid size`
The size of the given array is negative or exceeds memory size.

`attempt to extend common block <name> backwards`
An attempt has been made to extend a COMMON block backwards by means of EQUIVALENCE statements.

`bad length for CHARACTER value`
A value which is not positive has been used for a CHARACTER length.

`<class> storage block containing <name> is too large`
<class> is local or COMMON. The storage block containing the named variable exceeds memory size.

`concatenation too long`
The result of a CHARACTER concatenation may exceed memory size.

`conversion to integer failed`
A REAL or DOUBLE PRECISION value is too large for conversion to an integer.

`D to R real conversion failed`
A DOUBLE PRECISION value is too large for conversion to a REAL.

`DATA statement too complicated`
The variable list in a DATA statement is too complicated, and must be simplified.

`division by zero attempted in constant expression`
The divisor might be REAL, INTEGER, DOUBLE PRECISION or COMPLEX.

`real constant too large`
A REAL constant exceeds the permitted range.

`double constant too large`
A DOUBLE PRECISION constant exceeds the permitted range.

`inconsistent equivalencing involving <name>`
The given variable is involved in inconsistent EQUIVALENCE statements.

`increment in DATA implied DO-loop is zero`
A DATA statement implied DO loop has a zero increment.

`insufficient store for code generation`
The code generator has run out of workspace - the program unit being compiled must be simplified.

`insufficient values in DATA constant list`
There are more variables than constants in a DATA statement.

`integer invalid for length or size`
A value which is not positive has been used for a CHARACTER length or array size.

`lower bound exceeds upper bound in substring`
In a substring, a constant lower bound exceeds the constant upper bound.

`lower bound of substring is less than one`
A constant substring lower bound is less than one.

`upper bound exceeds length in substring`
A constant substring upper bound exceeds the length of the character variable.

`stack overflow - program must be simplified`
The internal expression stack has overflowed - the offending statement must be simplified.

`subscript below lower bound in dimension N`
A constant array subscript is less than the lower bound in the given dimension.

`subscript exceeds upper bound in the dimension N`
A constant array subscript exceeds the upper bound in the given dimension.

`too many constants in DATA statement`
There are more constants than variables in the DATA statement.

`too many program units in compilation`
The module limit must be increased.

`type mismatch in DATA statement`
The type of the constant is illegal for the corresponding variable.

`variable initialised more than once in DATA`
A variable has been initialised more than once by DATA statements in this program unit.

`wrong number of hex bytes for constant of TYPE type`
A hex constant has been given with the wrong number of digits.

`zero increment in DO-loop`
A DO loop with a constant zero increment value has been used.

`inconsistent use of NAME`
The external subroutine or function NAME has been used with inconsistent argument types. This error message would occur with the following program:

```
call abc(1.0)
call abc(2)
end
```

APPENDIX B

RUN-TIME ERROR MESSAGES

Code 1000 errors

`<ch> edit descriptor cannot handle logical list item`
Format descriptor used with a LOGICAL list item is not L; <ch> is the actual descriptor used.

`<ch> edit descriptor cannot handle character list item`
Format descriptor used with a CHARACTER list item is not A; <ch> is the actual descriptor used.

`<ch> edit descriptor cannot handle numeric list item`
Invalid descriptor for numeric value; <ch>is the actual descriptor used.

`Z field width unsuitable`
Wrong number of digits in hex (Z) input field for given type.

`FORMAT - unexpected character <ch>`
Invalid character <ch> in FORMAT.

`FORMAT - bad numeric descriptor`
Bad syntax for numeric FORMAT descriptor.

`FORMAT - cannot use when reading`
Quoted string used in input FORMAT.

`FORMAT - unexpected format end`
End of FORMAT inside quoted string.

`FORMAT - cannot use H when reading`
nH used in input FORMAT.

`FORMAT - bad scale factor`
Bad +nP or −nP construct.

`FORMAT - too many opening parentheses`
More than 20 nested opening parentheses (including the first).

`FORMAT - trouble with reversion`
No value has been or written by the repeated part of the format (this would cause an infinite loop if not trapped). The following program fragment illustrates the trouble with reversion format error:

```
write (1, 10) i, j
10 format (i5, (1x))
```

`FORMAT - width missing or zero`
Bad width in numeric edit descriptor.

`Unformatted output too long`
Unformatted record length exceeds maximum permitted. This can occur with direct access output only.

`Unformatted input record too short`
Input record does not contain sufficient data.

`mismatched use of ACCESS, RECL in OPEN`
ACCESS='DIRECT' has been quoted in an OPEN which does not contain a RECL specifier, or vice versa.

INPUT/OUTPUT ERRORS

1001 `invalid unit number`
Unit number not in range 1-60.

1002 `invalid attribute`
Invalid attribute used in OPEN statement

1003 `duplicate use of filename`
The same filename has been used more than once in an OPEN statement.

1004 `invalid unit for operation`
BACKSPACE/REWIND/ENDFILE attempted on unit connected for direct access.

1005 `error detected previously`
An IO error has been detected previously on this unit, and trapped with ERR=.

1006 `direct access without OPEN`
A direct access READ or WRITE has been used without an OPEN statement for the unit.

1007 `invalid use of unit`
Inconsistent use of unit (formatted mixed with unformatted, sequential mixed with direct access or ENDFILE done previously).

1008 `input and output mixed`
Input and output mixed on a sequential unit (without intervening REWIND or OPEN).

1009 `direct access not open for input`
The direct access file could not be opened for input (for example, file is write only).

1010 `direct access not open for output`
The direct access file could not be opened for output (for example, file is read only).

1011 `end of file on output`
An attempt has been made to write off the end of a sequential file. (In practice, this will occur with internal files only.)

1020 `invalid logical in input`
Formatted input file D contains bad logical value.

1021 `invalid number in input`
Bad number (range or syntax) in formatted I, D, E, F, or G input.

1022 `Bad complex data`
Bad COMPLEX constant in list directed input.

1023 `LD repeat not integer`
Repeat count (r*) in list directed input is not valid.

1024 `LD input data not REAL`
Syntax or range error in REAL list directed input value.

1025 `LD input data not INTEGER`
Syntax or range error in INTEGER list directed input value.

1026 `LD input data not DP`
Syntax or range error in DOUBLE PRECISION list directed input value.

1027 `LD input data not LOGICAL`
Syntax error in LOGICAL list directed input value.

1028 `LD input data not COMPLEX`
Syntax or range error in COMPLEX list directed input value.

1029 `LD input data not CHARACTER`
Syntax error in CHARACTER list directed input value.

1030 `LD repeat split CHARACTER`
Attempt to split a repeated character constant across a record boundary. This is strictly legal, but almost impossible to implement correctly.

2000 `not available`
BACKSPACE operation is not available.

2001 `bad unformatted record (message)`
A record in an unformatted file does not have the required structure.

2002 `invalid access to terminal file (message)`
Attempt to use terminal (or other output device) as an unformatted or direct access file. More detail is given.

2003 `sequential open failed (message)`
The actual reason for the failure (for example, Bad name) is given in the brackets.

2004 `direct access open failed (message)`
The actual reason for the failure (for example, Bad name) is given in the brackets.

2005 `direct access IO failed (message)`
For example, attempt to read past end of file.

2006 `record length too large`
The record length specified in a formatted direct access OPEN exceeds the permitted maximum (512 bytes).

2007 `bad direct access file (message)`
A file used for direct access has invalid initial data or insufficient record length.

2009 `bad command line syntax`

2010 `sequential write failed (message)`
I/O error on sequential output (for example, Can't extend)

APPENDIX C

FORTRAN C COMMAND

The C command is used to execute a command sequence with parameter substitution. Syntax:

```
C filename args...
```

The filename refers to the command sequence. If the file cannot be found in the current directory, then the directory $.execlib is tried. Under Arthur, the value of the system variable F77$Execlib (if set) is used instead of the default directory. The string should include a final dot.

If the C program is executed under a different name, that name is used for the command sequence, and no filename is read from the command line. For example, if a copy of the C program is placed in $.library.f77, then the command f77 will cause the command sequence $.execlib.f77 to be obeyed.

The C program functions by copying the command sequence to the temporary file $.tmp.exec, obeying directives and performing parameter substitution in the process. If the current input is from an exec file, this is appended to the temporary file, so that nested command sequences may be used. Finally the temporary file is executed by the command:

```
exec $.tmp.exec
```

Under Arthur, the value of the system variable F77$Tmp (if set) is used instead as the directory for the exec file. The string should include a final dot.

Values for F77$Execlib and F77$Tmp corresponding to the defaults are $.execlib. and $.tmp.

DIRECTIVES

A line in a command sequence that starts with a dot is a *directive.*

The possible directives are:

`.key keys` Defines the key string to be used to decode the parameters. The string is a series of keyword names, separated by commas. Each keyword may be followed by one or more of the following attributes:

/A:Argument is compulsory.

/K:If argument is present, keyword must be given.

/S:Keyword is a *switch* with no value.

A `.key` directive must be present if the sequence contains any parameter substitution; more than one `.key` is not allowed. `.k` is a synonym for `.key`.

Example:

```
.key from/a,to/a/k,opt/k,quick/s
```

In this example, the arguments with keywords from and to are compulsory; the keyword to must always be given. The argument opt is optional, but the keyword must be present if the argument is used. The keyword quick is an optional switch. Examples of valid command lines (assuming that the command file is called `comm`) are:

```
comm file1 -to file2
comm -from file1 -opt abc -to file2
comm -to file4 file3 -quick
```

`.dot char` Subsequent directives are introduced by char rather than dot.

Example:

```
.dot +
+key from/a,to/a/k
```

`.bra char` Set the opening bracket for parameter substitution to char. The initial value is <.

`.ket char` Set the closing bracket for parameter to char. The initial value is >.

`.dollar char` Set the character used to introduce parameter defaults to char. The initial value is $.

`.default key val` Set the default value of the keyword key to val. Only one `.default` directive is allowed for a keyword, and it must occur *after* the `.key` directive.

Example:

```
.default to vdu:
```

`.def` is a synonym for `.default`

`.concat char` Concatenate a non-directive line ending in char with the following line. There is no default concatenation character.

`.id title` Identification string. The string (with leading spaces removed) is output to the screen if the keystring contains an ID keyword and `-id` is quoted on the command line. Note that a `.id` directive must *precede* the `.key` directive.

`.help info` Help information line. Several `.help` lines are allowed; the information (preceded by the identification string, if any) is written to the screen if the key string contains a HELP keyword and -help is quoted on the command line. After writing the help text, the C command terminates. A single leading

space is removed from each line of help text. Note that all `.help` directives must precede the `.key` directive.

Directives with a space directly after the dot are ignored; they can therefore be used as comment lines.

PARAMETER SUBSTITUTION

The value of a parameter may be inserted by a reference like:

```
<key>
```

The bracket characters may be changed by `.bra` and `.ket` directives – see above. The value of a set switch (/S) parameter is its name.

A default value for the key may be specified:

```
<key$default>
```

The default may itself be a parameter reference:

```
<key$<key2$default>
```

or

```
<key$<key2$<key3$default>
```

etc.

The dollar character may be changed by the `.dollar` directive. The default is used if the key has not been set on the command line or by a `.default` directive. The key name must be in the key string or must be one of the following built-in names.

=DATE	The date in the form DD-MMM-YY.
=TIME	The time in the form HH:MM:SS.
=DAYNO	The day of the month in the form DD.
=MONTH	The date in the form MMM.
=YEAR	The date in the form 19YY.
=TMP	The standard temporary directory name (possibly set from F77$Tmp), with final dot.

If the system date has not been set, "<unset>" is used.

Under Arthur, a key name may also be a system variable, such as SYS$Time or F77$Execlib. A \ must be used to escape the $ character in such names (otherwise it would be treated as the default value prefix).

For example:

```
echo <Sys\$Time> <F77\$Execlib>
```

EXAMPLE

The following is an example command sequence to compile a FORTRAN program:

```
. Command sequence to compile FORTRAN
.
.id F77 command sequence version 1.00
.help
.help Keywords
.help
.help -from      Source file in directory f77
.help -object    Output file name in aof; default <from>
.help -opt       Compiler options
.help
.help For example:
.help
.help  f77 abc                compile f77.abc to aof.abc
.help  f77 prog -object x     compile f77.prog to aof.x
.help
.help Options:
.help
.help B     bound checking      Ln     line number level
.help 6     Fortran 66          7      strict Fortran 77
.help T     Tracing             Wn     warning level
.help Xn    Cross reference
.help
.help Default are: L1W2X0-BT67
.key from/a,object/k,opt/k,id/s,help/s,id=identify/s
f77fe f77.<from> -to <=tmp>fcode -opt <opt$+>
f77cg <=tmp>fcode -to aof.<object$<from> -opt <opt$+>
remove <=tmp>fcode
```

Note that ID and HELP keywords are included in the key string.

APPENDIX D

CALLING ASSEMBLER FROM FORTRAN

This section provides some guidance on the writing of assembler routines to be called from FORTRAN. It is assumed that the reader is familiar with ObjAsm.

Register conventions

FORTRAN programs use a simplified form of the standard Acorn calling conventions (as defined in Appendix C of the *Archimedes Programmer's Reference Manual*). The main differences are that only one argument is passed in all calls, and that the registers reserved for register variables (v1-v6 and f4-f7) are not preserved by FORTRAN subprograms. Thus C programs cannot call FORTRAN (because registers are not preserved), although FORTRAN can call C.

The detailed register usage is:

R0-R9	Scratch registers.
FP (R10)	Used to refer to argument list within subprogram.
SP (R12)	Standard run-time stack.
SB (R13)	Used to refer to local data within subprogram.

Argument lists

Every call in FORTRAN passes *one* argument in R0 (A1). This is a pointer to a list of the addresses of the arguments given in the call (every argument in FORTRAN is passed by reference). Thus the address of the first argument is at [R0,#0], the second at [R0,#4], etc. If an assembler routine does not call any other routines, the argument list can be left in R0. Normally, however, the address of the list is copied to FP (R10), which is preserved by calls.

For a CHARACTER argument, the address in the argument list does not refer directly to the data; instead it points at a *character descriptor*, which is an eight-byte block containing the address of the character value in its first word and its length in the second. For example, if the third argument in a call is a character value, the following code fragment loads its address into R1 and its length into R2:

```
LDR    R1,[R0,#8]     ; Descriptor address
LDMIA  R1,{R1,R2}     ;Address and length
```

Function results

For non-CHARACTER functions, the address of the result is passed back in R0 (ie. the result is stored and the address of the location loaded into R0). A

character function is implemented as a subroutine with the address of the result location (as a character descriptor) passed as an extra *first* argument (thus the first argument in the call appears as the second argument, and so on).

A subroutine with alternate returns (*'s in the argument list) is implemented as an INTEGER function. The result should be zero for the main return, one for the first alternate return, two for the second, etc. The alternate return specifiers do not appear in the argument list.

Static data

Static data for an assembler routine should be allocated in a writeable area and addressed using SB (R13) within the routine (this register is preserved by calls).

Section format

The code area in a FORTRAN assembler module should start with the routine name as a twelve-character string, padded with spaces. The address of the first byte after this name is pushed to the stack during entry. The code area should be named *F77$$Code* and have attributes CODE and READONLY. The data area (if any) should be named *F77$$Data* and have the DATA attribute.

The basic layout of a FORTRAN assembler section is:

```
        TTL     "name"

; Registers

R0      RN       0
R1      RN       1

        ...
        ...
R9      RN       9
FP      RN      10
SP      RN      12
SB      RN      13
R14     RN      14
PC      RN      15

F0      FN       0
        ...
        ...
F7      FN       7
```

```
; Data
        AREA    |F77$$Data|,DATA
        ... data declarations ...
; Code
        AREA    |F77$$Code|,CODE,READONLY
NAME    DCB     "modname     "
DATAPTR DCD     |F77$$Data|        ; Address of data
        ... code ...
        END
```

COMMON blocks and NOINIT

FORTRAN common blocks should be defined as named AREAs with the COMMON and NOINIT attributes. An initialised COMMON block (equivalent to a BLOCK DATA subprogram) should be defined with the COMDEF (common definition) attribute. FORTRAN blank common is given the name F77_BLANK.

Entry and exit sequences

The standard entry sequence for a FORTRAN-callable routine is:

```
        ADR     R1,NAME+12
        STMFD   SP!,{R1,FP,SB,R14}
        LDR     SB,DATAPTR          ; Address data area
        MOV     FP,R0               ; Copy argument list
```

NAME refers to the twelve character module name at the start of the section. The return sequence is:

```
        LDMFD   SP!,{R1,FP,SB,PC}
```

A complete example

This (rather contrived) example illustrates the use of function results and CHARACTER arguments. The FORTRAN 77 equivalent of ICHSUM would be:

```
        INTEGER FUNCTION ICHSUM(CH, ARRAY)
        COMMON /CMAX/ MAX
        CHARACTER *(*) CH
        INTEGER ARRAY(*)
```

```
ICHSUM = 0
MAX    = 0
DO J = 1,LEN(CH)
  ICH = ICHAR(CH(J:J))
   ARRAY(J) = ICH
   ICHSUM = ICHSUM + ICH
   IF (ICH .GT. MAX) MAX = ICH
END DO
END
```

In other words, the characters from the string are assigned as integers into the array, and the sum of all the characters is returned as the result. The maximum value in the array is assigned to a variable in COMMON block *CMAX*. The assembler version of ICHSUM might be:

```
        TTL     "ichsum"
; Registers
R0      RN       0
R1      RN       1
R2      RN       2
R3      RN       3
R4      RN       4
R5      RN       5
R6      RN       6
R7      RN       7
R8      RN       8
R9      RN       9

FP      RN      10
SP      RN      12
SB      RN      13
R14     RN      14
PC      RN      15

F0      FN       0
F1      FN       1
F2      FN       2
F3      FN       3
F4      FN       4
F5      FN       5
F6      FN       6
F7      FN       7
```

```
; Data

        AREA    |F77$$Data|,DATA
RESULT  %       4               ; Result location

; Common blck

        AREA    CMAX,DATA,COMMON,NOINIT
        %       4

; Code

        AREA    |F77$$Code|,CODE,READONLY

NAME    DCB     "ICHSUM      "
DATAPTR DCD     |F77$$Data|

        EXPORT  ICHSUM

; INTEGER FUNCTION ICHSUM(CH, IARRAY)

ICHSUM  ADR     R1,NAME+12
        STMFD    SP!,{R1,FP,SB,R14}
        LDR     SB,DATAPTR
        MOV     FP,R0

        LDMIA   FP,{R0,R2}      ; Argument addresses
        LDMIA   R0,{R0,R1}      ; CH address and length

        MOV     R3,#0           ; Initialise sum
        MOV     R5,#0           ; And maximum

LOOP    LDRB    R4,[R0],#1      ; Next character
        STR     R4,[R2],#4      ; Store in array
        ADD     R3,R3,R4        ; Add to sum

        CMP     R5,R4           ; Compare with maximum
        MOVLT   R5,R4

        SUBS    R1,R1,#1        ; Reduce count
        BGT     LOOP            ; Round again

        LDR     R0,COMMPTR      ; Get COMMON address
        STR     R5, [R0]         ; Store maximum

        STR     R3,[SB]         ; Store result
        MOV     R0,SB           ; Address of result

        LDMFD   SP!,{R1,FP,SB,PC}

COMMPTR DCD     CMAX            ; Address of common block
        END
```

A sample FORTRAN program which uses ICHSUM might be:

```
INTEGER X(10)
COMMON /CMAX/ MAX
ISUM = ICHSUM('0123456789', X)
PRINT *, ISUM, MAX, X
END
```

Assuming that the assembler text and FORTRAN test program have been typed into the files `asm.ichsum` and `f77.test` respectively, they could be compiled and linked as follows:

```
objasm asm.ichsum aof.ichsum -quit -stamp
f77 test
link aof.test aof.ichsum $.libarary.lib.f77/1 -output test
```

APPENDIX E

CALLING C FROM FORTRAN

FORTRAN programs can also call routines written in C. The C procedure has to accept a single argument which refers to a list of addresses. A simple way of doing this is to declare the argument as a pointer to a structure containing pointer fields. Functions have to be declared as returning a pointer result. As an example, a C version of the assembler above might be:

```
typedef struct {char * address; int length;} chardesc;
static int sum;

int * ICHSUM (struct {chardesc * ch; int * array;} * args)
  { int j;

    sum = 0;

    for (j = 0; j <args -> ch -> length; j ++)
      { int ich = (args -> ch -> address) [j];

        sum                    +=ich;
        (args -> array) [j]    =ich;
      }
    return & sum;
  }
```

This C function has the same specification as the assembler routine except that the maximum is not returned in the COMMON variable (C cannot access FORTRAN COMMON blocks). Note that the function name must be given in upper case. The C procedure could be compiled and linked as follows:

```
cc -c ichsumc
link aof.test o.ichsumc $.library.lib.f77/1
                $.arm.clib.ansilib/1 -image test
```

The link command must be typed on one line. Note that both the FORTRAN and the C libraries must be given.

APPENDIX F

SAMPLE ASD SESSION

This section gives a sample FORTRAN debugging session using ASD, the Arthur Symbolic Debugger. The example program is necessarily very simple and the fault it contains is easy to spot, but it should serve as an introduction to ASD.

The following program is supposed to print out the date of Easter Sunday for the years 1970-1989 (using a very arcane algorithm). Note that the line numbers are not part of the program, but are included to enable references to be made in the text below.

```
 1      PROGRAM EDATE
 2      INTEGER Y, DAY
 3      CHARACTER *(5) MONTH
 4
 5      DO Y = 1970,1989
 6        CALL EASTER DATE(Y, DAY, MONTH)
 7      PRINT '(I4, 1X, I2, 1X, A)', Y, DAY, MONTH
 8      END DO
 9      END
10
11      SUBROUTINE EASTER DATE(YEAR, DAY, MONTH)
12      INTEGER YEAR, DAY, G, C, X, Z, D, E, N
13      CHARACTER *(*) MONTH
14
15      G = MOD(YEAR, 19) + 1
16      C = YEAR / 100 + 1
17      X = (3 * C) / 4 - 12
18      Z = (8 * C + 5) / 25 - 5
19      D = (5 * YER) / 4 - X - 10
20      E = MOD (11 * G + 20 + Z - X, 30)
21
22      IF (E .EQ. 25 .AND. G .GT. 11 .OR. E .EQ. 24) THEN
23        E = E + 1
24      ENDIF
25
26      N = 44 - E
27      IF (N .LT. 21) N = N + 30
28      N = N + 7 - MOD(D+N, 7)
29
30      IF (N .GT. 31) THEN
31        DAY   = N - 31
32        MONTH= 'April'
```

```
33      ELSE
34        DAY   = N
35        MONTH = 'March'
36      ENDIF
37      END
```

If this program is entered exactly as above and run, the results at first sight seem correct – the date for 1988 (3 April) is right. However, the date shown for 1989 is incorrect – the true result is 26 March. The result for 1970 is also wrong – it should be 29 March.

To use ASD to investigate this program, it is first compiled and linked with debug information included:

```
f77 edate –debug all
linkf77 edate
```

It is assumed that the source of the program is in f77.edate. To enter the debugger, use the following command:

```
*asd edate
```

The first step might be to check that the results are printed correctly. So a break point is set on the PRINT statement in the main program (line 7):

```
ASD: brk edate:7
```

`edate` in the brk command is the main program name; if the program had not started with a PROGRAM statement, the default name F77_MAIN would have been used instead. Note that upper and lower case letters in names are not distinguished when debugging FORTRAN programs, so the brk command above could have been typed in any of the following forms:

`ASD: BRK EDATE:7` or
`ASD: brk Edate:7` or
`ASD: brk EdAtE:7` or
`ASD: BRK eDaTe:7` etc.

The program is run until the PRINT statement is reached and the value of DAY is displayed:

```
ASD: go
Program stopped as breakpoint #1, location edate:7
ASD: print day
27
```

The value 27 shows that the fault lies in EASTER DATE – the correct result is 29. The next step is to trace though the subroutine:

```
ASD: brk easterdate
ASD: go
Program stopped as breakpoint #2, location easterdate
ASD: print year
1971
```

The print shows that the argument (YEAR) is correct. A simple step might be to set a break point after the initial calculations, on line 22:

```
ASD: brk 22
ASD: go
Program stopped as breakpoint #3, location 22
ASD: print g
15
ASD: print c
20
ASD: print x
3
ASD: print z
1
ASD: print d
-13
ASD: print e
3
```

The value of D (−13) looks a bit suspicious, so the value of the expression is displayed:

```
ASD: print (5*year/4-x-10
2450
```

2450 is not the same as −13! Inspection of the expression for D (line 19) shows that YEAR was mistyped as YER; D was therefore computed using an undefined value. To check whether this is the only problem, D can be set to the correct value (it is not used in an expression until line 28) and execution resumed:

```
ASD: let d=2450
ASD: unbreak #1
ASD: go
1971 11 April
Program stopped as breakpoint #2, location easterdate
```

The breakpoint on line 7 was cleared so that a result would be printed without stopping. The new result for 1971 is correct. If the program is altered and recompiled, the results for each year are now right.

This sample session has given a very superficial introduction to the powerful facilities available in ASD, which include procedure tracing and variable watchpoints. The reader is referred to the ASD reference manual for more information.

INDEX